Improving Discrete Mathematics and Algorithms Curricula with *LINK*

Jonathan Berry
Elon College.
berryj@numen.elon.edu

Abstract

This paper introduces the *LINK* system as an educational tool which can be used to visualize and experiment with discrete algorithms. An extended example demonstrates the flexibility of the system in the context of a fundamental graph algorithm: finding the strongly connected components of a directed graph.

1 Background

A major factor governing the success of collaborative learning initiatives in computer science and discrete mathematics is the meaningful use of technology which students can control themselves. Thorough, active investigations of fundamental topics is an attractive idea, but one that comes at the cost of reduced material coverage. Educational software should be expected both to give students quick intuition in areas which are not treated thoroughly in class, and to enable them to investigate selected topics in more detail than would be possible with paper and pencil.

Discrete mathematical branches of computer science and mathematics are especially good candidates for the use of such software since so many topics in data structures, algorithms, automata, and discrete structures are readily visualized. However, although the need for discrete mathematics software is now reflected in the fact that both Maple and Mathematica have graph theory and combinatorics libraries, no system has provided an interface sufficient to allow students to take true advantage of the visual nature of discrete mathematics. Students should be able to edit graphs and their attributes using a mouse-driven GUI, program experiments and animations using a computing environment such as those of Maple and Mathematica, and implement algorithms and property tests using object-oriented libraries which spare them most low-level programming chores.

Over the past several years, there have been several efforts to design software systems for discrete mathematics to meet these goals. None, however, has resulted in a product with influence comparable to the now-familiar symbolic mathematics packages.

A lack of consensus on the part of discrete mathematics researchers regarding the ideal nature of such a system and a lack of entrepreneurial support to assume the burden of quality software development have contributed to this shortcoming.

Some notable existing systems for discrete mathematics are *Combinatorica* [10], Steven Skiena's extension package for *Mathematica*, *Netpad* [7] due to Nathaniel Dean and others at Bellcore, *SetPlayer* [1], due to Mark Goldberg and his students at Rensselaer Polytechnic Institute, and Gregory Shannon et. al.'s *GraphLab* [9]. For various reasons, none of these systems has the potential to be a widely-useful environment for both graph manipulation and computation. The authors of these systems recognized this, and along with Daniel Gorenstein, the founding director of DIMACS, proposed and obtained funding for *LINK*, which was to be a freely-available and portable software system for discrete mathematics which would various shortcomings of existing systems. After three years of development, the system is now freely available from the *LINK* web site:

`http://dimacs.rutgers.edu/Projects/LINK.html.`

2 An extended example

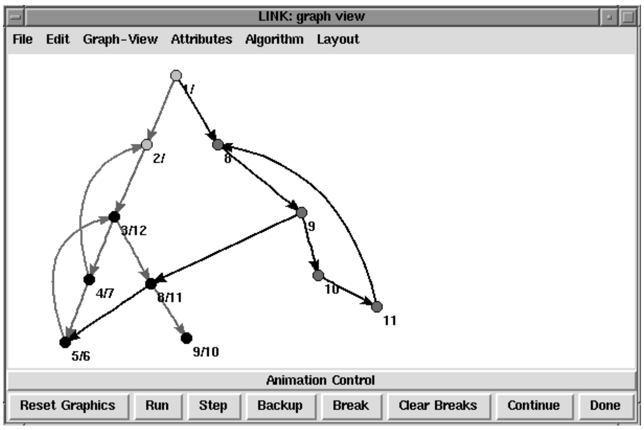

Figure 1: **DFS**

The following extended example provides some good illustrations of *LINK*'s potential as an educational tool. We will consider a problem found in the intersection of the studies of algorithms and discrete mathematics. Recall that any depth-first search of a directed graph associates with each vertex the first and last times it is encountered during the search. These times are known as the *discovery* and *finishing* times. *LINK*'s animation controller can be used to show search in progress, as well as to step through it both backwards and forwards, to set breakpoints, rerun at any time, and continue to the next breakpoint. While the algorithm is running, labels identifying each vertex's discovery and finishing times and each edge's classification (tree, back, forward, cross) can be displayed or not in any combination. Figure 1 shows a search in progress.

1. Call depth-first search to compute the finishing times $f[u]$ for each vertex u in G.
2. Compute G^T, the *transpose* of G, where $G^T = (V, E^T)$, and $E^T = \{(u, v) : (v, u) \in E(G)\}$.
3. Call depth-first search on G^T, but in the main loop of the algorithm, consider the vertices in order of decreasing $f[u]$ (as computed in line 1).
4. Output the vertices of each tree in the depth-first forest of step 3 as a separate strongly connected component.

Figure 2: **An algorithm for finding strongly connected components**

Consider the algorithm in Figure 2, which finds the *strongly connected components* [1] of a directed graph $G = (V, E)$ given in Corman, Leiserson, & Rivest's well-known *Introduction to Algorithms*(CLR) [3].

```
(define (random-walk graph-view)
   (do ((v (random-vertex (graph graph-view))
           (random-neighbor v)))
       ((not v) (reset-graphics graph-view))
       (flash (vertex-item v graph-view))))
```

Figure 3: **A random walk function**

As Corman, Leiserson, and Rivest state, this remarkable algorithm seems to have nothing to do with strongly connected components (let us simply call them *components*). The remainder of this section will consider a series of assertions which support the claim that the algorithm is correct and show ways in which *LINK*'s features can be used to explain the assertions. Descriptions of visual lessons will be accompanied by the *LINK* code which realizes them.

2.1 Random walks and component layouts

The first assertion proved in CLR is that no path between two vertices in the same component ever leaves that component. The proof is not difficult, but even before a proof is presented, performing a series of random walks through the graph will make it clear to students that once we leave a component, we cannot return. To visualize a random walk, we need to select a random vertex and identify it somehow, pick a random neighbor of that vertex and identify it, and repeat this process as long as possible (which might mean forever). Figure 3 contains a sample piece of *LINK* code for taking random walks.

This code, written in *LINK*'s command language, [2] assigns to the variable v a randomly-selected vertex of the given "graph-view," or window containing a graph. The "vertex-item" of v, a special screen object corresponding to the vertex v, is then retrieved and flashed on the screen. A random neighbor is then found (if one exists), and the process is repeated as long as the current vertex has a neighbor. The intuition behind the assertion is effectively demonstrated by using *LINK*'s built-in "component layout" to arrange the vertices into their respective components (see Figure 8), then undertaking several random walks. If doubts persist, the instructor can use the mouse to add a directed edge back into a component from an outside vertex reachable from that component. Further random walks will show that it is now possible to leave and reenter the "component," but selecting the component layout from the layout menu once more shows that the addition of the edge has changed the components, and the walk never left and returned to a component after all.

2.2 Forefathers and key bindings

The *forefather* $\phi(v)$ of a vertex v with respect to a depth-first search is a difficult concept for some students to grasp. Formally defined, $\phi(v)$ is the vertex w such that there is a path from v to w and w's finishing time is maximized. This concept is one of the keys to understanding strongly connected components, and *LINK* can be used quite easily to bring the concept to life. It is a simple matter to bind a key on the computer's keyboard so that clicking on a vertex with the mouse, then hitting that key will flash [3] the forefather of the selected vertex. The *LINK* code is shown in its entirety in Figure 4. When invoked, this code calls a strongly connected components algorithm which sets the *forefather* attribute [4] in each vertex v to be $\phi(v)$. Subsequently, when the "f" key is pressed after a vertex has been selected, the *forefather* attribute (a vertex) is retrieved, and the corresponding vertex-item in the graph window is found and flashed. In fact, this key binding makes it possible to present the concept as a puzzle without first defining it. With the graph's vertices arranged with *LINK*'s random layout option, repeated clicking and highlighting seems to reveal no pattern. When the experiment is repeated with the vertices laid out by *LINK*'s depth-first layout, however, patterns appear and students start making conjectures. A typical conjecture for the meaning of the flashed vertex might be: "The vertex

[1] A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $U \subseteq V$ such that there is a path from each vertex $v \in U$ to every other vertex in U.

[2] *LINK*'s interface is written in STk, Erik Gallesio's Scheme interface to John Ousterhout's Tk graphics package [8][5]

[3] or change the vertex's color, size, border, etc.

[4] This attribute is built-in, but new attributes can be added on a temporary basis interactively, or on a permanent basis by recompiling the system.

```
(define (forefather-binding graph-view)
 (strongly-connected-components (graph graph-view))
 (bind (slot-ref graph-view 'graph-toplevel) "<KeyPress-f>"
      (lambda (x y)
       (flash (vertex-item
                 (find-vertex-attribute 'forefather
                    (slot-ref (car *link:selected-vertex-items*) 'vertex)) graph-view)))))
```

Figure 4: **Binding the "f" key so that the forefather of a selected vertex is flashed.**

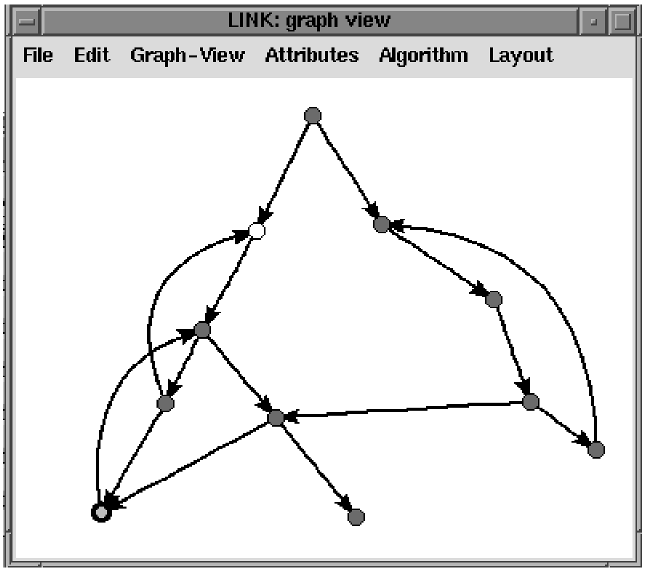

Figure 5: **Forefather finding as a puzzle**

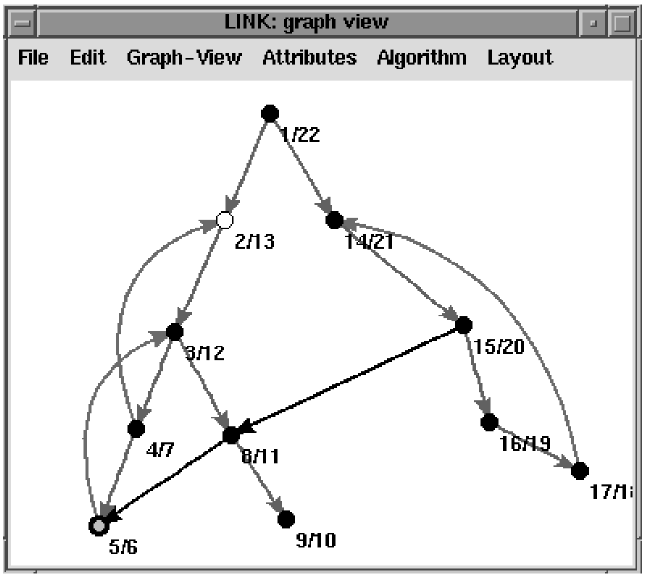

Figure 6: **Forefather finding with finishing times depicted**

obtained by following a path down as far as possible, then following one edge back up." If the student conjectures are incorrect, counterexamples can be shown visually simply by clicking and highlighting the true forefather. Figure 5 shows a counterexample to the preceding conjecture.

Once the concept of forefather has been clarified, several assertions can be made clear by once again selecting the component layout option. When the vertices of each component are located together, two assertions stated in CLR become intuitively clear: every vertex in a component has the same forefather and the forefather of the forefather of a vertex is the forefather of that vertex. Figure 8 shows the graph in its component layout with each forefather enlarged. The forefather test shown in Figures 5 and 6 is being repeated with clearer results.

2.3 The depth-first search of G^T

What about the algorithm for strongly connected components in Figure 2? It is probably still not clear why it works. *LINK* has already been able to shed some light on the key elements of the correctness proof, but there is more it can do to lend intuition and engage students. Figure 7 shows the main *LINK* window along with a graph window containing G, the menu of graph operations selected from the main window, and a window containing G^T which was brought up by selecting "transpose." The excerpt of STk code which produces the menu is shown in Figure 9. *LINK*'s interface provides functions such as *unary-operation* and *binary-predicate*, which automate the process of prompting the user to click in a graph window, running the operation, and displaying any results.

In the new graph window containing G^T, we can visualize step 3 of the algorithm. When G^T was created, its vertices and edges inherited all of the attributes of their alter-egos in G, including *finishing time*. This attribute can now be used to drive the necessary depth-first search of G^T. *LINK*'s depth-first search routine is written in such a way that customizations are straightforward, and adding the resulting menu option to the interface is a short process. Figure 10 shows the current search options.

Selecting the depth-first search by decreasing finishing time for G^T groups the strongly connected components into a forest of disjoint depth-first search trees. The depth-first search of G^T in Figure 7 has the property that a set of vertices connected by tree edges comprises a strongly connected component. In the figure, the tree edges have been thickened to illustrate this property. The enlarged vertices are the roots of their respective depth-first search trees and therefore the forefathers of the graph. The thickened edges in G^T show that when the vertices are ordered by discovery time in the depth-first search of G^T, the vertices of each component are contiguous in that order.

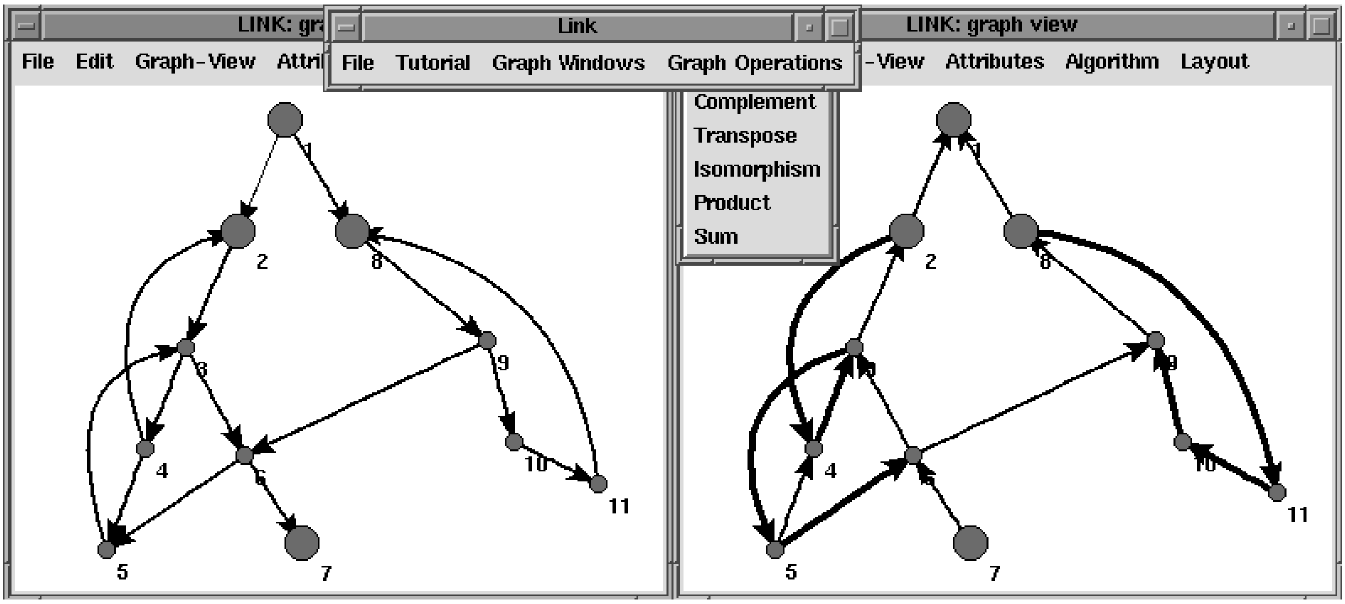

Figure 7: **Windows containing G and G^T**

Even after visualizing this algorithm and key concepts involved with its correctness proof, it may still not be intuitively clear why it works. *LINK* is designed to be flexible enough to allow a professor or student who has spent some time learning Scheme and the design of *LINK*'s interface to respond to new questions with new visualizations.

3 Student Experimentation

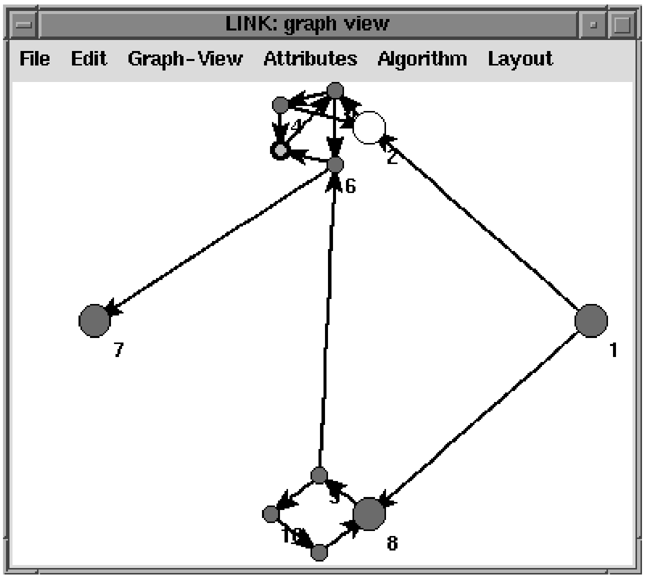

Figure 8: **Forefather finding with the component layout**

The preceding section considered *LINK* as a professor's tool, but clearly, it can be useful and fun for computer science students. The only reason that the student view is not the primary emphasis of this paper is that the current documentation is not modular or complete enough to effectively teach the system to novice computer scientists in a very short period of time. It should be stressed, however, that the system *is* usable by students and that I am committed to improving both it and the documentation. Currently, the user's and programmer's manuals total over 170 pages and include dozens of examples.

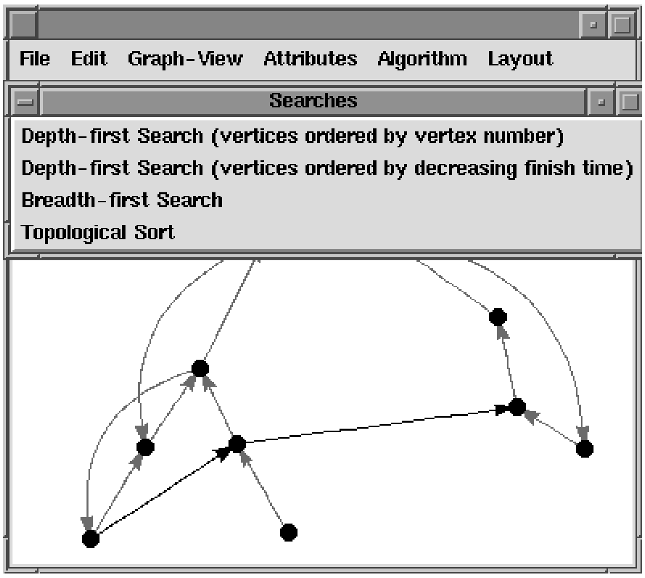

Figure 10: **The search algorithms menu**

LINK was designed to be a tool for experimentation with discrete mathematical concepts, and effective experimentation implies thoroughness. The command language has been included to make the design and implementation of experiments easy. As an example,

```
("Graph Operations"
  ("Complement"  ,(lambda() (unary-operation complement #t)))
  ("Transpose"   ,(lambda() (unary-operation transpose #t)))
  ("Isomorphism" ,(lambda() (binary-predicate isomorphic?
     "nauty: They are isomorphic"
     "nauty: They are not isomorphic")))
  ("Product" ,(lambda() (binary-operation product #t)))
  ("Sum" ,(lambda() (binary-operation sum #t)))
)
```

Figure 9: ***LINK*'s extendible *Graph Operations* menu**

consider Figure 12, which depicts a short experiment: the generation of three random graphs for which minimum spanning trees are computed and displayed.

More advanced computer science students will be able to use *LINK*'s object-oriented libraries to program their own algorithms, animations, and experiments. *LINK* provides class libraries similar in spirit to the LEDA [6] class libraries, but more focused on discrete mathematical objects such as graphs and hypergraphs. It is straightforward to compile a program using *LINK*'s libraries without any interface, and the process of incorporating new methods into the STk interface is not hard, though it and its documentation must be streamlined to be widely useful.

```
#include<iostream.h>
#include<LINK/graph/UHyperGraph.h>
main()
{
        UHyperGraph g;
        Set<Vertex*> edge1, edge2;
        Vertex *v1, *v2, *v3;

        v1 = g.addVertex("v1");
        v2 = g.addVertex("v2");
        v3 = g.addVertex("v3");
        edge1.insert(v1);
        edge1.insert(v2);
        edge2.insert(v1);
        edge2.insert(v2);
        edge2.insert(v3);
        g.addEdge(edge1);
        g.addEdge(edge2);
        cout << g << endl;
}
```

```
{[v1 v2 v3]  {{v1 v2} {v1 v2 v3}}}
```

Figure 11: **A simple *LINK* program and its output**

4 Design Philosophy

LINK was designed for discrete mathematics research and education, not for processing massive data sets. Issues such as flexibility, extensibility, and portability took clear precedence over speed, and decisions were made accordingly. The interpretive STk graphics system is much slower than a pure Windows or X/Motif application, but for *LINK*'s anticipated users, the benefits of extendible graphics, Scheme command language, and portability far outweigh the cost in speed. A study of the experiences of secondary school teachers using Netpad and Combinatorica especially pointed out the need for flexibility and extensibility in educational applications [4].

Currently, *LINK* runs only on Unix systems (including Linux), but there is no major obstacle preventing a port to Windows, since the STk graphics system has already been ported. *LINK* for Windows might become a reality before the end of 1997. Although a port to the Macintosh is more problematic (since STk has not yet been ported there), portability remains an important priority and concern, and we are committed to making *LINK* more accessible in the future.

The current system is described in the manual available from the web site, and the latter also contains links to a preview paper and a paper concerning *LINK* and market basket analysis. The early development of *LINK* is detailed in [2].

There is still much more work to be done to make *LINK* a widely useful tool. The algorithms library lacks numerous fundamental components, and there are no course materials designed to make use of *LINK*. DIMACS, Elon College, and RPI are actively seeking funding to improve this situation by preparing an extensive, modular set of packages of web-based course materials intended to help teach various topics in discrete mathematics and algorithms using *LINK* as a primary tool.

I plan to organize a DIMACS workshop on using *LINK* once it is available for Windows.

Acknowledgments

I would like to acknowledge the support of DIMACS and the *LINK* grant: CCR-9214487. DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Laboratories, Lucent Technologies/Bell Laboratories Innovations, and Bellcore. DIMACS is an NSF Science and Technology Center, funded under contract STC-91-19999; and also receives support from the New Jersey Commission on Science and Technology. I would also like to acknowledge the contributions of *LINK* primary investigators: Nathaniel Dean, Mark Goldberg, Gregory Shannon, and Steven Skiena, the original project leader, Patricia Fasel of Los Alamos National Laboratory, and the many students who have helped with the *LINK* effort.

References

[1] BERQUE, D., CECCHINI, R., GOLDBERG, M., AND RIVENBURGH, R. The setplayer system for symbolic computation on power sets. *Journal of Symbolic Computation 14* (1992), 645–662.

[2] BERRY, J., DEAN, N., FASEL, P., GOLDBERG, M., JOHNSON, E., MACCUISH, J., SHANNON, G., AND SKIENA, S. LINK: A combinatorics and graph theory workbench for applications and research. Tech. Rep. 95-15, Center for Discrete Mathematics and Theoretical Computer Science (see also: http://dimacs.rutgers.edu), Piscataway, NJ, 1995.

[3] CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction to Algorithms*. McGraw-Hill, 1990.

[4] DEAN, N., AND LIU, Y. Discrete mathematics software for K–12 education. In *Discrete Mathematics in the Schools: Making an Impact, D. Franzblau, F. Roberts, and J. Rosenstein, eds.* (To appear), American Mathematical Society, DIMACS series.

[5] GALLESIO, E. The STk reference manual. Tech. Rep. RT 95-31a, I3S CNRS, Université de Nice - Sophia Antipolis, France, 1995.

[6] MEHLHORN, K., AND NÄHGER, S. Leda: A platform for combinatorial and geometric computing. *CACM 38*, 1 (Jan 1995), 96–102.

[7] MEVENKAMP, M., DEAN, N., AND MONMA, C. NETPAD user's guide and reference guide, 1990.

[8] OUSTERHOUT, J. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[9] SHANNON, G., MEEDEN, L., AND FRIEDMAN, D. SchemeGraphs: An object-oriented environment for manipulating graphs, 1990. Software and documentation.

[10] SKIENA, S. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley, 1990.

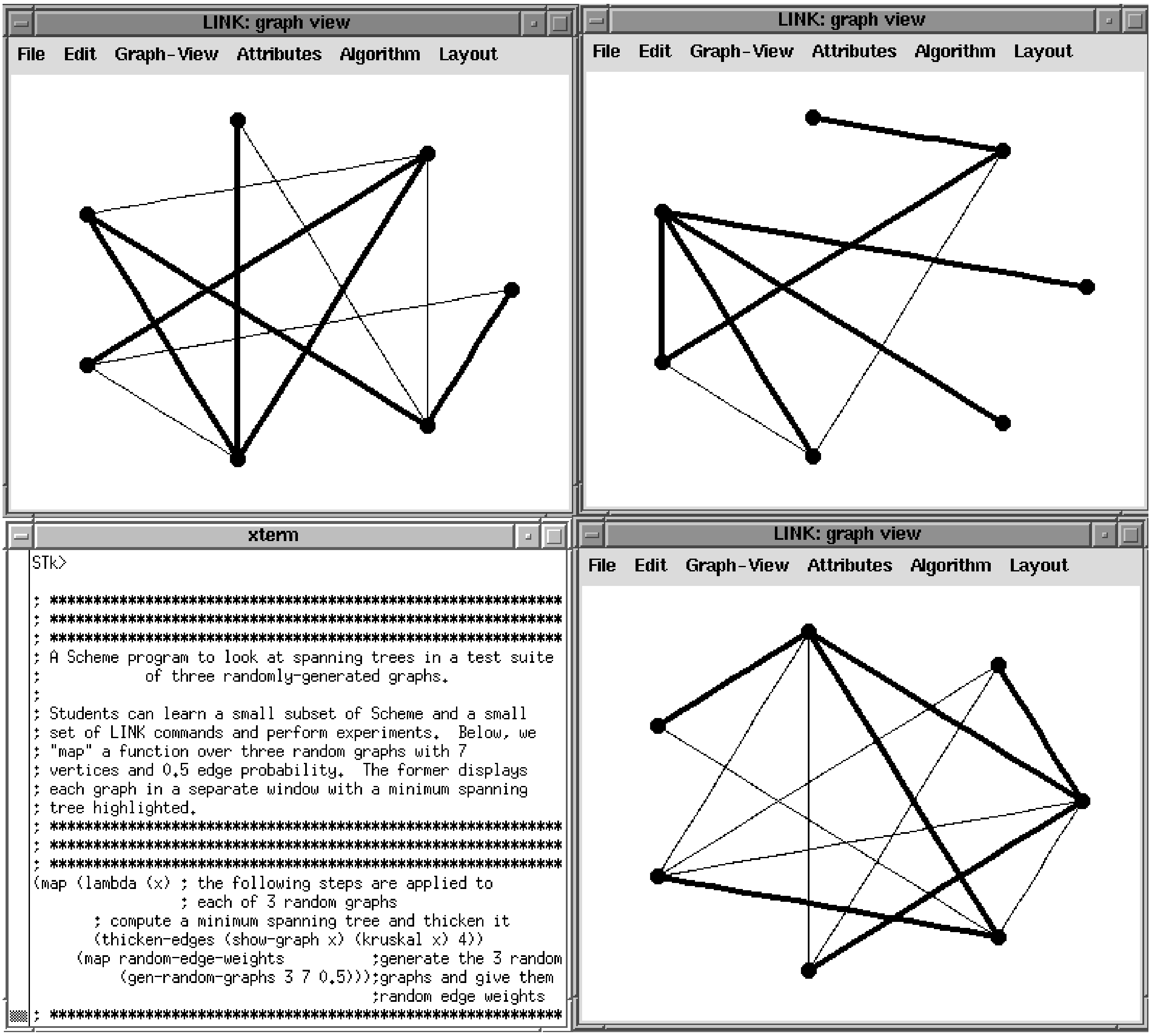

Figure 12: **A *LINK* program implementing an experiment with minimum spanning trees of random graphs**